Understanding the classical model for linking: You can override an LIB with another LIB, and a LIB with an OBJ, but you can't override an OBJ

devblogs.microsoft.com/oldnewthing/20130109-00

January 9, 2013

Raymond Chen

If you study the classical model for linking, you'll see that OBJ files provided directly to the linker have a special property: They are added to the module *even if nobody requests a symbol from them*.

OBJs bundled into a library are pulled into the module only if they are needed to resolve a *needed* symbol request. If nobody needs a symbol in the OBJ, then the OBJ doesn't get added to the module. On the other hand, OBJs handed directly to the linker get added to the module *whether anybody wants them or not*.

Last time, we learned about the *along for the ride* technique which lets you pull components into a module even if they were not explicitly requested by an OBJ. Today's problem is sort of the reverse of this: If you move an OBJ from the explicit OBJ list to a library, then somebody has to remember to take it for a ride.

Some time ago, Larry Osterman described how some components use sections to have one component automatically register itself with another component when the OBJ is pulled into the module. But in order for that to work, you have to make sure the OBJ gets pulled into the module in the first place. (That's what Larry's `CallForceLoad` function is for: By putting it an explicit OBJ, that function forces the OBJ from the LIB to be pulled in. And then, since nobody ever calls `CallForceLoad`, a later linker pass discards it as an unused function.)

Another consequence of the algorithm by which the linker pulls OBJs from libraries to form a module is that if a *needed* symbol can be satsified without consulting a library, then the OBJ in the library will not be used. This lets you override a symbol in a library by explicitly placing it an OBJ. You can also override a symbol in a library to putting it in *another* library that gets searched ahead of the one you want to override. But you can't override a symbol in an explicit OBJ, because those are part of the initial conditions.

Exercise:

Discuss this user's analysis of a linker issue.

> I have three files:
>
> ```
> // awesome1.cpp
> int index;
>
> // awesome2.cpp
> extern int index;
> void setawesomeindex(int i)
> {
>  index = i;
> }
>
> // main.cpp
> int index = 0;
> int main(int, char**)
> {
>  setawesomeindex(3);
>  return index;
> }
> ```
>
> When I link the object files together, I get an error complaining that `index` is multiply defined, as expected. On the other hand, if I put `awesome1.cpp` and `awesome2.cpp` into a library, then the program links fine, but the two copies of the `index` variable were merged by the linker! When I set the awesome index to 3, it also changes my main program's variable `index` which has the same name. Why is the linker merging my variables, and how can I keep them separate?
>
> When I share my `awesome.lib` with others, I don't want to have to give them a list of all my global variables and say, "Don't create a global variable with any of these names, because they will conflict with my library." (And that would also prevent me from adding any new global variables to my library.)

Exercise: Clarify the following remark by making it more precise and calling out the cases where it is false. "Multiple definitions for a symbol are allowed if they appear in LIBs."

Exercise (harder): The `printf` function is in a bit of a pickle regarding whether it should support the floating point formats. If it includes them unconditionally, then its use of the floating point data types causes the floating point emulation library to be linked into the module, even if the module didn't otherwise use floating point! Use what you've learned so far this week to provide one way that the `printf` function could determine whether it should include floating point format support based on whether the module uses floating point.

Raymond Chen

Follow